**Внутренние компоненты ASP.NET Core 1.0**

**Обзор**

ASP.NET Core 1.0 представляет собой ASP.NET с существенными изменениями. Например, в ASP.NETCore вы составляете конвейер запросов с использованием связующего программного обеспечения. Связующее ПО ASP.NET Core использует асинхронную логику обработки на HttpContext, а затем при необходимости применяет следующее связующее ПО в цепочке или непосредственно прекращает запрос. Как правило, вы *Используете* связующее ПО путем применения соответствующего метода расширения на IApplicationBuilder в вашем методе Конфигурации. Примерами компонентов связующего ПО могут быть маршрутизация и аутентификация.

В данном модуле вы узнаете о некоторых программных компонентах ASP.NET Core 1.0, включая то, как работать с компонентами связующего ПО, и даже как создавать заказные настройки.

**Цели обзора**

В этом модуле вы узнаете, как:

* Работать со статическими файлами
* Работать с маршрутизацией
* Писать заказное связующее ПО
* Добавлять Аутентификацию к вашим веб-приложениям

**Требования**

Для завершения данного модуля потребуется следующее:

* [Visual Studio Community 2015 (бесплатно)](https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx) или выше
* [Visual Studio 2015 Update 2](http://go.microsoft.com/fwlink/?LinkId=691129)
* [.NET Core SDK](https://dot.net/) включая
  + [Visual Studio tools for .NET Core - Preview 1](https://go.microsoft.com/fwlink/?LinkId=798481) (для поддержки Visual Studio)
  + [NuGet Manager extension](https://dist.nuget.org/visualstudio-2015-vsix/v3.5.0-beta/NuGet.Tools.vsix) версия 3.5 или более поздняя
  + [.NET Core SDK for Windows](https://go.microsoft.com/fwlink/?LinkID=798398) (для поддержки командной строки)

**Примечание:** воспользуйтесь подпиской на [Visual Studio Dev Essentials](https://www.visualstudio.com/en-us/products/visual-studio-dev-essentials-vs.aspx), чтобы получить все, что нужно для создания и развертывания вашего приложения на любой платформе.

**Установка**

Для выполнения упражнения в этом модуле, вам необходимо сначала выполнить следующие настройки.

1. Откройте Windows Explorer и перейдите к папке модуля **Source**.
2. Щелкните правой кнопкой мыши **Setup.cmd** и выберите **Запуск от имени администратора**, чтобы запустить процесс установки, который будет настраивать среду и устанавливать фрагменты кода Visual Studio для этого модуля.
3. Если вы видите диалоговое окно Контроль Учетных Записей Пользователей, подтвердите действие, чтобы продолжить.

**Примечание:** Убедитесь, что вы проверили все зависимости для этого модуля перед запуском установки.

**Использование фрагментов кода**

На протяжении всего документа модуля вам будут даны инструкции вставить блоки кода. Для вашего удобства, большая часть этого кода предоставляется фрагментам кода Visual Studio, к которым вы можете получить доступ в Visual Studio 2015, чтобы избежать необходимости добавлять их вручную.

**Примечание**: Каждое упражнение сопровождается стартовым решением, находящимся в папке упражнения **Begin**, что позволяет вам заниматься каждым упражнением независимо от других. Обратите внимание, что фрагменты кода, которые добавляются во время выполнения упражнений, отсутствуют в этих стартовых решениях и могут не работать до тех пор, пока вы не выполните упражнение. Внутри исходного кода для упражнения вы также найдете папку **End**, содержащую решение Visual Studio с кодом, который является результатом завершения действия, описанных в соответствующем упражнении. Вы можете использовать эти решения в качестве руководства, если вам потребуется дополнительная помощь во время работы с модулем.

**Упражнения**

Данный модуль включает в себя следующие упражнения:

1. [Работа со статическими файлами](https://github.com/Microsoft-Build-2016/CodeLabs-WebDev/tree/master/Module2-AspNetCore#Exercise1)
2. [Введение в маршрутизацию и систему контроля и управления MVC](https://github.com/Microsoft-Build-2016/CodeLabs-WebDev/tree/master/Module2-AspNetCore#Exercise2)
3. [Построение класса связующего ПО](https://github.com/Microsoft-Build-2016/CodeLabs-WebDev/tree/master/Module2-AspNetCore#Exercise3)
4. [Добавление Аутентификации в ваши веб-приложения](https://github.com/Microsoft-Build-2016/CodeLabs-WebDev/tree/master/Module2-AspNetCore#Exercise4)

Примерное время для завершения работы по данному модулю: **60 минут**

При первом запуске Visual Studio необходимо выбрать одну из предопределенных наборов настроек. Каждый заранее определенный набор разработан так, чтобы соответствовать определенному стилю разработки, и он определяет схемы размещения окон, поведение редактора, фрагменты кода IntelliSense и опции диалоговых окон. Процедуры, описанные в данном модуле, характеризуют действия, необходимые для выполнения данной задачи в Visual Studio при использовании **General Development Settings (Стандартных Параметров Разработки)**. Если выберете другие параметры для среды разработки, обратите внимание, что при этом могут возникнуть различия в действиях.

**Упражнение 1: Работа со статическими файлами**

Статические файлы, которые включают в себя HTML и CSS файлы, файлы изображений, JavaScript файлы, являются активами, которые приложение отправляет напрямую клиенту.

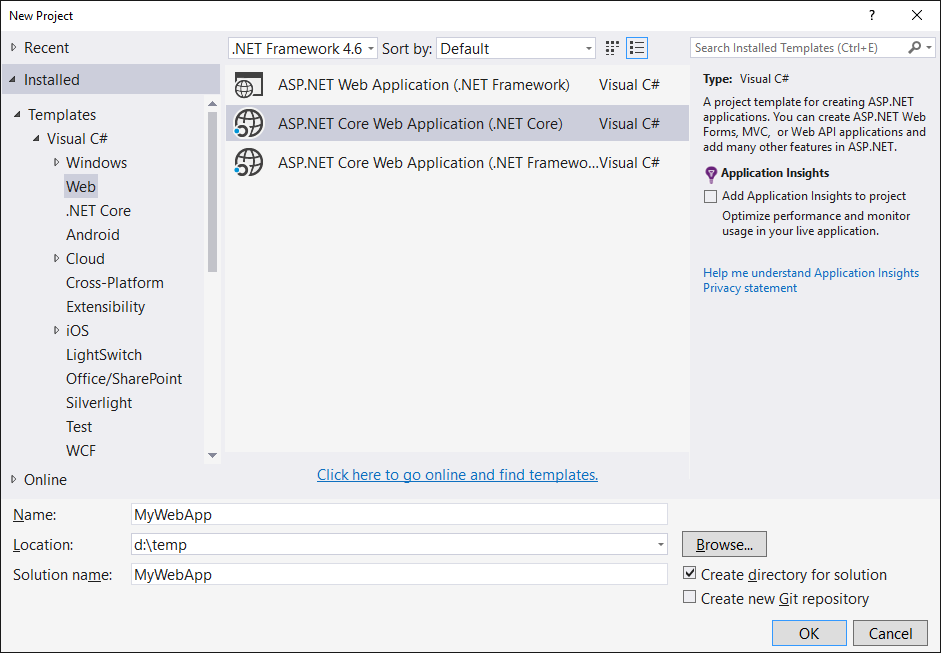
В этом упражнении вы настроите свой проект так, чтобы предоставлять статические файлы.

**Задание 1 – Предоставление статических файлов**

Для того, чтобы предоставить статические файлы, вы должны сконфигурировать связующее ПО, чтобы добавить статические файлы к конвейеру обработки. Это достигается путем вызова метода расширения **UseStaticFiles** из метода **Startup.Configure**.

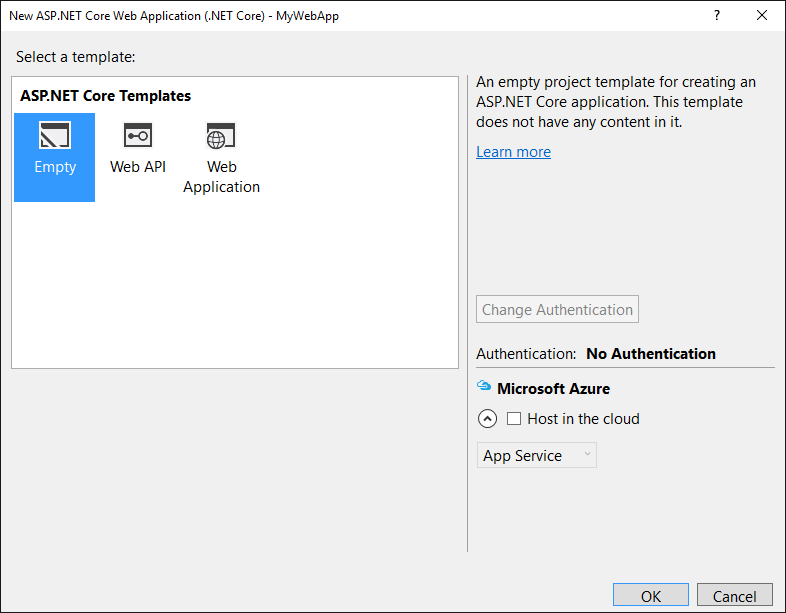
В этой задаче вы создадите пустой проект ASP.NET Core 1.0 и настроите его для обработки статических файлов.

1. Откройте **Visual Studio Community 2015** и выберите **File | New Project...** для начала работы.
2. В диалоговом окне **New Project** выберите **ASP.NET Web Application** во вкладке **Visual C# | Web**, убедитесь, что выбрано **.NET Framework 4.6**. Назовите проект *MyWebApp*, выберите **Location** и нажмите **OK**.



*Создание нового проекта веб-приложения ASP.NET*

1. В диалоговом окне **New ASP.NET Project** выберите шаблон **Empty**. Нажмите **OK**.



*Создание нового проекта при помощи пустого шаблона ASP.NET Core*

1. Добавьте пакет **Microsoft.AspNet.StaticFiles** как зависимость к **project.json**.

"dependencies": {

"Microsoft.AspNetCore.Server.IISIntegration": "1.0.0-rc2-final",

"Microsoft.AspNetCore.Server.Kestrel": "1.0.0-rc2-final",

"Microsoft.AspNetCore.StaticFiles": "1.0.0-rc2-final"

},

1. Откройте файл **Startup.cs** и добавьте вызов метода **UseStaticFiles** в метод **Configure** до hello world связующего ПО.

public void Configure(IApplicationBuilder app)

{

app.UseStaticFiles();

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World!");

});

}

1. Создайте файл с названием **index.html** со следующим содержимым в папке **wwwroot**.

(Code Snippet - *ASPNETCore - Ex1 - IndexHtml*)

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Hello static world!</title>

</head>

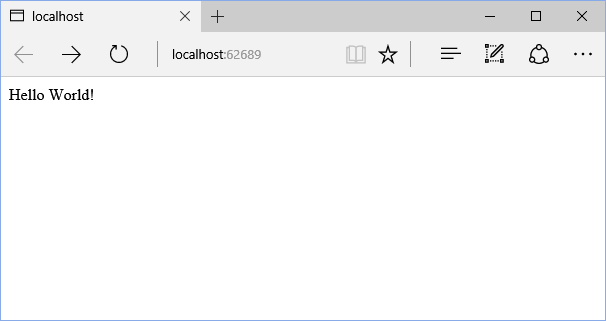
<body>

<h1>Hello from ASP.NET Core!</h1>

</body>

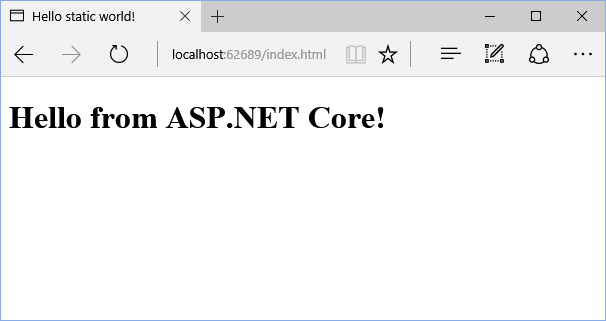
</html>

1. Запустите приложение и перейдите к корню. Он должен показать hello world связующего программного обеспечения.

[](https://github.com/Microsoft-Build-2016/CodeLabs-WebDev/blob/master/Module2-AspNetCore/Images/hello-world.png?raw=true)

*Hello World*

1. Перейдите к **index.html.** Вы должны видеть статическую страницу в **wwwroot:**

[](https://github.com/Microsoft-Build-2016/CodeLabs-WebDev/blob/master/Module2-AspNetCore/Images/hello-world-aspnet-core.png?raw=true)

*Статическая страница Hello World*

**Задание 2 – Добавление поддержки документа по умолчанию**

Для того, чтобы ваше веб-приложение поддерживало страницу по умолчанию без того, чтобы пользователю приходилось полностью определять URI, существует метод расширения **UseDefaultFiles**. Этот метод *переписывает* URL, который не показывает файл.

В дополнение к методам расширений **UseStaticFiles** и **UseDefaultFiles**, есть также один метод - **UseFileServer** - который сочетает в себе функциональные возможности этих двух методов, а также методы расширений **UseDirectoryBrowser**.

В этом задании вы будете использовать **UseFileServer** для работы как со статическими файлами, так и с файлами по умолчанию.

1. Откройте файл **Startup.cs** и измените статические файлы связующего ПО в методе **Configure** из app.UseStaticFiles() в app.UseFileServer().

public void Configure(IApplicationBuilder app)

{

app.UseFileServer();

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World!");

});

}

1. Запустите приложение снова. Страница по умолчанию **index.html** должна отображаться при переходе к корню сайта.

**Упражнение 2: Введение в маршрутизацию и MVC**

Маршрут представляет собой шаблон URL, который отображается в обработчике. Обработчиком может быть физический файл, такой как файл .aspx в приложении Web Forms. Обработчик также может быть классом, который обрабатывает запрос, такой как контроллер в приложении MVC.

Маршрутизация ASP.NET позволяет использовать URL-адреса, которые не имеют для сопоставления определенных файлов на веб-узле. Поскольку URL не должен отображаться в файл, вы можете использовать URL-адреса, которые описывают действия пользователя и, следовательно, более понятны для пользователей.

В этом упражнении вы узнаете, как настроить маршрутизацию в вашем приложении.

**Задание 1 - Добавление MVC**

ASP.NET MVC дает вам мощный, шаблонный способ создания динамических веб-сайтов, что позволяет четко разделить проблемы и что дает вам полный контроль над разметкой для приятной, гибкой разработки. ASP.NET MVC включает в себя множество функций, которые позволяют быстро осуществлять разработку посредством тестирования для создания сложных приложений, использующих самые последние веб-стандарты.

В этом задании вы будете настраивать проект для использования ASP.NET MVC и конфигурации выборочного маршрута.

1. Откройте **Visual Studio Community 2015** и решение **MyWebApp.sln,** которое находится в папке **Source/Ex1/End**. В качестве альтернативы, вы можете продолжить с решением, которое вы получили в предыдущем упражнении.
2. Откройте файл **project.json** и добавьте **Microsoft.AspNet.Mvc** в раздел **dependencies**.

"dependencies": {

"Microsoft.NETCore.App": {

"version": "1.0.0-rc2-3002702",

"type": "platform"

},

"Microsoft.AspNetCore.Server.IISIntegration": "1.0.0-rc2-final",

"Microsoft.AspNetCore.Server.Kestrel": "1.0.0-rc2-final",

"Microsoft.AspNetCore.StaticFiles": "1.0.0-rc2-final",

"Microsoft.AspNetCore.Mvc": "1.0.0-rc2-final"

},

1. В **Solution Explorer** кликните правой кнопкой мыши по проекту **MyWebApp** и выберите **Add | New Folder**, назовите папку *Controllers*.
2. Кликните правой кнопкой мыши и выберите **Add | New Item...**, убедитесь, что узел **.NET Core** выбран слева, выберите **MVC Controller Class**, назовите файл *HomeController.cs* и кликните **Add**.
3. Замените содержимое файла с помощью следующего фрагмента кода.

(Code Snippet - *ASPNETCore - Ex2 - HomeController*)

using Microsoft.AspNetCore.Mvc;

namespace MyWebApp.Controllers

{

public class HomeController : Controller

{

// GET: /<controller>/

public string Index() => "Hello from MVC!";

}

}

1. Теперь откройте файл **Startup.cs** и добавьте сервисы MVC и связующее ПО к конфигурации, добавляя services.AddMvc() и замещая вызов метода app.Run в методе **Configure** методом UseMvc, как показано в следующем фрагменте кода.

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app)

{

app.UseFileServer();

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

}

1. Запустите веб-сайт и проверьте, что сообщение возвращается от MVC контроллера, перейдя к **/home**.

**Примечание:** ASP.NET Core MVC также включает в себя удобный новый служебный метод, app.UseMvcWithDefaultRoute(), так что вам не нужно запоминать шаблон строки.

**Задание 3: Написание заказного связующего ПО**

Небольшие компоненты приложений, которые могут быть включены в конвейер запросов HTTP, известны как связующее программное обеспечение. ASP.NET 1.0 Core имеет встроенную поддержку для связующего программного обеспечения, которое собрано в приложении методом **Configure** во время *Application Startup*.

В этом упражнении вы создадите класс связующего ПО, который устанавливает текущие параметры, основанные на значении строки запроса.

**Примечание:** Мы используем связующее ПО, относящееся к локализации, в этом упражнении в качестве примера сценария, но в большинстве случаев вы будете использовать [ASP.NET Core's built-in support localization support](https://docs.asp.net/en/latest/fundamentals/localization.html).

**Задание 1 – Написание класса связующего ПО, которое задает текущую культуру на основе значения строки запроса.**

Связующее ПО - это компоненты, которые собраны в поток, чтобы обрабатывать запросы и ответы. Каждый компонент может передавать запрос следующему компоненту в потоке и выполнять определенные действия до и после вызова следующего компонента. Делегаты запросов используются для создания этих потоков, которые служат затем для обработки каждого HTTP запроса.

Делегаты запросов настраиваются с помощью методов расширения **Run, Map** и **Use** для **IApplicationBuilder**, который передается в метод **Configure** класса **Startup**. Отдельный делегат запроса может быть указан как анонимный метод либо же может быть определен в многократно используемом классе. Этими многократно используемыми классами является связующее ПО или его компоненты. Каждый компонент связующего ПО в потоке запросов отвечает за вызов следующего компонента в цепочке, либо обходит несколько компонентов, чтобы достичь нужного.

В этом задании вы создадите встроенное связующее ПО.

1. Откройте **Visual Studio Community 2015** и выберите **File | New Project...** для создания нового решения, используя шаблон **ASP.NET Web Application**, назовите его *MiddlewareApp*, кликните **OK** и затем выберите шаблон **Empty** в **ASP.NET 5 Templates**.
2. Откройте файл **Startup.cs** и замените содержимое метода **Configure** следующим фрагментом кода, который создает встроенное связующее программное обеспечение, которое запускается **до** делегата hello world, который задает культуру для текущего запроса из строки запроса.

(Code Snippet - *ASPNETCore - Ex3 - InlineMiddleware*)

public void Configure(IApplicationBuilder app)

{

app.Use((context, next) =>

{

var cultureQuery = context.Request.Query["culture"];

if (!string.IsNullOrWhiteSpace(cultureQuery))

{

var culture = new CultureInfo(cultureQuery);

CultureInfo.CurrentCulture = culture;

CultureInfo.CurrentUICulture = culture;

}

// Call the next delegate/middleware in the pipeline

return next();

});

app.Run(async (context) =>

{

await context.Response.WriteAsync($"Hello {CultureInfo.CurrentCulture.DisplayName}");

});

}

1. Используйте решения для отсутствующих параметров.
2. Загрузите приложение, чтобы увидеть связующее программное обеспечение в действии, установите культуру, добавив строку запроса **culture**, например, [*http://localhost:64165/?culture=no*](http://localhost:64165/?culture=no)

**Задание 2 - Перемещение связующего ПО к своему типу**

В данном задании вы переместите связующее ПО в отдельный файл.

1. Кликните правой кнопкой мыши по проекту **MiddlewareApp** и выберите **Add | Class...**, далее назовите файл *RequestCultureMiddleware.cs* и кликните **Add**.
2. Добавьте конструктор, который принимает параметр **RequestDelegate**, и назначьте его в частной области, используя следующий фрагмент кода. Используйте решения для отсутствующих параметров, когда требуется.

(Code Snippet - *ASPNETCore - Ex3 - RequestCultureMiddlewareClass*)

public class RequestCultureMiddleware

{

private readonly RequestDelegate next;

public RequestCultureMiddleware(RequestDelegate next)

{

this.next = next;

}

}

1. Добавьте следующий метод с содержанием встроенного связующего программного обеспечения делегата, ранее добавленного в файл **Startup.cs**.

(Code Snippet - *ASPNETCore - Ex3 - RequestCultureMiddlewareInvokeMethod*)

public Task Invoke(HttpContext context)

{

var cultureQuery = context.Request.Query["culture"];

if (!string.IsNullOrWhiteSpace(cultureQuery))

{

var culture = new CultureInfo(cultureQuery);

CultureInfo.CurrentCulture = culture;

CultureInfo.CurrentUICulture = culture;

}

// Call the next delegate/middleware in the pipeline

return this.next(context);

}

1. В нижней части файла добавьте класс, который предоставляет связующее программное обеспечение с помощью метода расширения на **IApplicationBuilder**.

(Code Snippet - *ASPNETCore - Ex3 - RequestCultureMiddlewareExtensionsClass*)

public static class RequestCultureMiddlewareExtensions

{

public static IApplicationBuilder UseRequestCulture(this IApplicationBuilder builder)

{

return builder.UseMiddleware<RequestCultureMiddleware>();

}

}

1. Еще в файле приложения **Startup.cs**, замените делегат связующего программного обеспечения вызовом метода app.UseRequestCulture(), чтобы добавить новый класс связующего программного обеспечения к конвейеру HTTP. Когда вы закончите, ваш метод настройки должен выглядеть как код ниже:

public void Configure(IApplicationBuilder app)

{

app.UseRequestCulture();

app.Run(async (context) =>

{

await context.Response.WriteAsync($"Hello {CultureInfo.CurrentCulture.DisplayName}");

});

}

1. Запустите приложение и убедитесь, что связующее программное обеспечение работает как класс.

**Задание 3 – Добавление параметров в связующее ПО**

В этом задании вы будете обновлять выполнение **RequestCultureMiddleware** для поддержки установления культуры по умолчанию в качестве значения конфигурации.

1. Кликните правой кнопкой мыши по проекту **MiddlewareApp** и выберите **Add | Class...**, назовите файл *RequestCultureOptions.cs* и кликните **Add**.
2. В новом классе добавьте свойство **DefaultCulture** с **CultureInfo** как тип, разрешая недостающую зависимость.

public class RequestCultureOptions

{

public CultureInfo DefaultCulture { get; set; }

}

1. Откройте файл *RequestCultureMiddleware.cs* и обновите конструктор **RequestCultureMiddleware**, выбрав параметр **RequestCultureOptions,** как это показано в следующем фрагменте кода.

public class RequestCultureMiddleware

{

private readonly RequestDelegate next;

private readonly RequestCultureOptions options;

public RequestCultureMiddleware(RequestDelegate next, RequestCultureOptions options)

{

this.next = next;

this.options = options;

}

//...

}

1. Обновите метод **Invoke** связующего ПО, чтобы использовать свойство **DefaultCulture** из **options,** если ничего не было указано в строке запроса, как показано в следующем фрагменте кода.

(Code Snippet - *ASPNETCore - Ex3 - InvokeMethodWithDefaultCulture*)

public Task Invoke(HttpContext context)

{

CultureInfo requestCulture = null;

var cultureQuery = context.Request.Query["culture"];

if (!string.IsNullOrWhiteSpace(cultureQuery))

{

requestCulture = new CultureInfo(cultureQuery);

}

else

{

requestCulture = this.options.DefaultCulture;

}

if (requestCulture != null)

{

CultureInfo.CurrentCulture = requestCulture;

CultureInfo.CurrentUICulture = requestCulture;

}

return this.next(context);

}

1. В том же файле замените класс реализации **RequestCultureMiddlewareExtensions** на следующий фрагмент кода, который добавляет перегрузку к методу **UseRequestCulture**, который принимает **RequestCultureOptions** и передает его в вызов UseMiddleware<RequestCultureMiddleware>.

(Code Snippet - *ASPNETCore - Ex3 - UpdatedRequestCultureMiddlewareExtensions*)

public static IApplicationBuilder UseRequestCulture(this IApplicationBuilder builder)

{

return builder.UseRequestCulture(new RequestCultureOptions());

}

public static IApplicationBuilder UseRequestCulture(this IApplicationBuilder builder, RequestCultureOptions options)

{

return builder.UseMiddleware<RequestCultureMiddleware>(options);

}

1. Откройте файл *Startup.cs* и установите для запасного варианта культуры в методе **Configure** значение по умолчанию в виде, например, *"en-GB"*.

(Code Snippet - *ASPNETCore - Ex3 - UpdatedUseRequestCulture*)

app.UseRequestCulture(new RequestCultureOptions

{

DefaultCulture = new CultureInfo("en-GB")

});

1. Запустите приложение и убедитесь в том, что культура по умолчанию, если не указана строка запроса, соответствует сконфигурированному.

**Задание 4 - Чтение конфигурации культуры запроса из файла**

Система конфигурации ASP.NET Core была вновь спроектирована с предыдущих версий ASP.NET, которые полагались на **System.Configuration** и конфигурации XML-файлов, таких как **web.config**. Новая модель конфигурации обеспечивает прямой доступ к основным настройкам/значениям на основании данных, которые можно извлечь из различных поставщиков. Приложения и фреймворк затем могут получить доступ к настройкам параметров с помощью новой модели **Options**.

В этом задании вы будете использовать новую систему загрузки **Configuration,** загружающей значение культуры **RequestCultureOptions** по умолчанию из файла JSON.

1. Откройте файл *Startup.cs* и добавьте новую область приватного класса с именем **configuration** типа **IConfiguration**, разрешающие недостающую зависимость для **IConfiguration**.

public class Startup

{

private readonly IConfiguration configuration;

// ...

}

1. Добавьте новый конструктор, создайте новый объект **Configuration** в конструкторе при помощи **ConfigurationBuilder** и назначьте его на поле класса конфигурации, созданной на предыдущем шаге.

(Code Snippet - *ASPNETCore - Ex3 - StartupConstructor*)

public Startup(IHostingEnvironment env)

{

var configuration = new ConfigurationBuilder()

.Build();

this.configuration = configuration;

}

1. Откройте файл *project.json* и добавьте отсылку к пакету **Microsoft.Extensions.Configuration.Json** в узле **зависимостей**.

"dependencies": {

"Microsoft.NETCore.App": {

"version": "1.0.0-rc2-3002702",

"type": "platform"

},

"Microsoft.AspNetCore.Server.IISIntegration": "1.0.0-rc2-final",

"Microsoft.AspNetCore.Server.Kestrel": "1.0.0-rc2-final",

"Microsoft.Extensions.Configuration.Json": "1.0.0-rc2-final"

},

1. В файле *Startup.cs* добавьте вызов AddJsonFile("config.json") сразу после создания объекта **ConfigurationBuilder** в качестве цепного метода.

public class Startup

{

private readonly IConfiguration configuration;

public Startup(IHostingEnvironment env)

{

var configuration = new ConfigurationBuilder()

.SetBasePath(env.ContentRootPath)

.AddJsonFile("config.json")

.Build();

this.configuration = configuration;

}

// ...

}

1. Правой кнопкой мыши кликните проект **MiddlewareApp** и выберите **Add | New Item...**, выберите **JSON file** в качестве шаблона, назовите файл *config.json* и кликните **Add**.
2. В новом файле *config.json* добавьте новую пару ключ/значение "culture": "en-US".

{

"culture": "en-US"

}

1. Откройте файл *Startup.cs* и обновите код для того, чтобы задать культуру по умолчанию с использованием новой системы конфигурации.

app.UseRequestCulture(new RequestCultureOptions

{

DefaultCulture = new CultureInfo(this.configuration["culture"] ?? "en-GB")

});

1. Запустите приложение и проверьте, что культура по умолчанию является одним в заданных параметров в конфигурационном файле.
2. Обновите значения культуры "es" в файле *config.json* и обновите страницу (без изменения какой-либо другого кода). Обратите внимание, что сообщение не изменилось, так как конфигурация была только прочитана, когда приложение уже было запущено.
3. Вернитесь назад в Visual Studio и перезапустите веб сервер, нажав **Ctrl + Shift + F5**.
4. Вернитесь назад в браузер и обновите страницу; вы должны увидеть обновленное сообщение.

**Задание 5 – Параметры потоки системы от внедрения зависимостей до связующего ПО**

ASP.NET Core разработан, чтобы поддержать и максимально использовать добавление зависимости. Приложения ASP.NET Core могут использовать встроенные фреймворк сервисы, вводя их в методы в классе Startup, сервисы приложений могут быть также настроены для добавления. Контейнер сервисов по умолчанию, предоставленный ASP.NET Core, предоставляет собой минимальный набор функциональных возможностей и не предназначен для замены других контейнеров.

В этом задании вы будете использовать систему добавления зависимостей для настройки параметров **RequestCultureMiddleware**.

1. Откройте файл *project.json* и добавьте ссылку на пакет **Microsoft.Extensions.OptionsModel** в узле **зависимостей**.

"dependencies": {

"Microsoft.NETCore.App": {

"version": "1.0.0-rc2-3002702",

"type": "platform"

},

"Microsoft.AspNetCore.Server.IISIntegration": "1.0.0-rc2-final",

"Microsoft.AspNetCore.Server.Kestrel": "1.0.0-rc2-final",

"Microsoft.Extensions.Configuration.Json": "1.0.0-rc2-final",

"Microsoft.Extensions.Options": "1.0.0-rc2-final"

},

1. Измените конструктор **RequestCultureMiddleware** на IOptions<RequestCultureOptions> вместо RequestCultureOptions и получите значения параметров. Разрешите недостающие зависимости.

public RequestCultureMiddleware(RequestDelegate next, IOptions<RequestCultureOptions> options)

{

this.next = next;

options = options.Value;

}

1. Обновите класс **RequestCultureMiddlewareExtensions,** удаляя метод с вариантами параметра и вызывая UseMiddleware<RequestCultureMiddleware> в другом методе, как показано в следующем фрагменте кода.

public static class RequestCultureMiddlewareExtensions

{

public static IApplicationBuilder UseRequestCulture(this IApplicationBuilder builder)

{

return builder.UseMiddleware<RequestCultureMiddleware>();

}

}

1. В *Startup.cs* измените связующее ПО **UseRequestCulture** на непринятие никаких аргументов.

app.UseRequestCulture();

1. В методе **ConfigureServices**, расположенном в том же самом фале, добавьте строчку, которая конфигурирует культуру, используя метод services.Configure<RequestCultureOptions> и добавьте вызов метода **AddOptions**, как это показано в следующем фрагменте кода.

(Code Snippet - *ASPNETCore - Ex3 - ConfigureServices*)

public void ConfigureServices(IServiceCollection services)

{

// Setup options with DI

services.AddOptions();

services.Configure<RequestCultureOptions>(options =>

{

options.DefaultCulture = new CultureInfo(this.configuration["culture"] ?? "en-GB");

});

}

1. Запустите приложение и убедитесь, что параметры сконфигурированы из системы добавления зависимостей.

**Упражнение 4: Добавление Аутентификации в ваши веб-приложения**

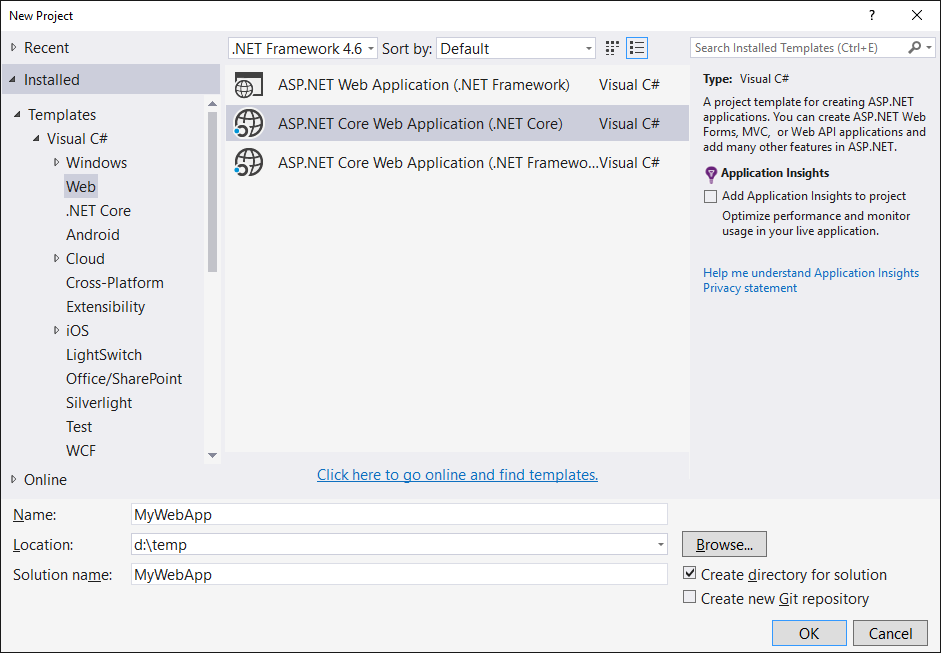
**ASP.NET Identity** представляет собой систему членства, которая позволяет добавлять функциональные возможности входа в приложение. Пользователи могут создать учетную запись и войти в систему с именем пользователя и паролем, или они могут использовать внешний поставщики входа в систему, такие как Facebook, Google, Microsoft Account, Twitter и многие другие.

В этом упражнении вы рассмотрите конфигурации по умолчанию шаблонов проектов ASP.NET Core, которые используют ASP.NET Identity, а также настроите **Facebook** в качестве внешнего поставщика авторизации в приложении.

**Задание 1 – Введение в ASP.NET Identity**

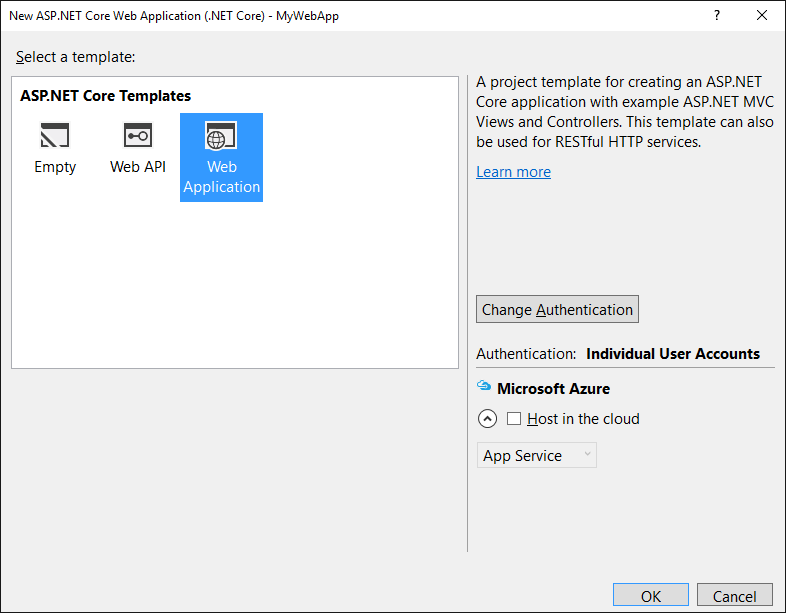
В этом задании вы узнаете, как шаблоны проектов ASP.NET Core используют ASP.NET Identity, чтобы добавить функциональность для регистрации, входа в систему и выхода из системы пользователя.

1. Откройте **Visual Studio Community 2015** и выберите **File | New | Project...** для создания нового решения.
2. В диалоговом окне **New Project** выберите **ASP.NET Web Application** во вкладке **Visual C# | Web** и убедитесь, что выбран **.NET Framework 4.6**. Дайте проекту название, например, *MyWebApp*, выберите **Location** и нажмите **OK**.



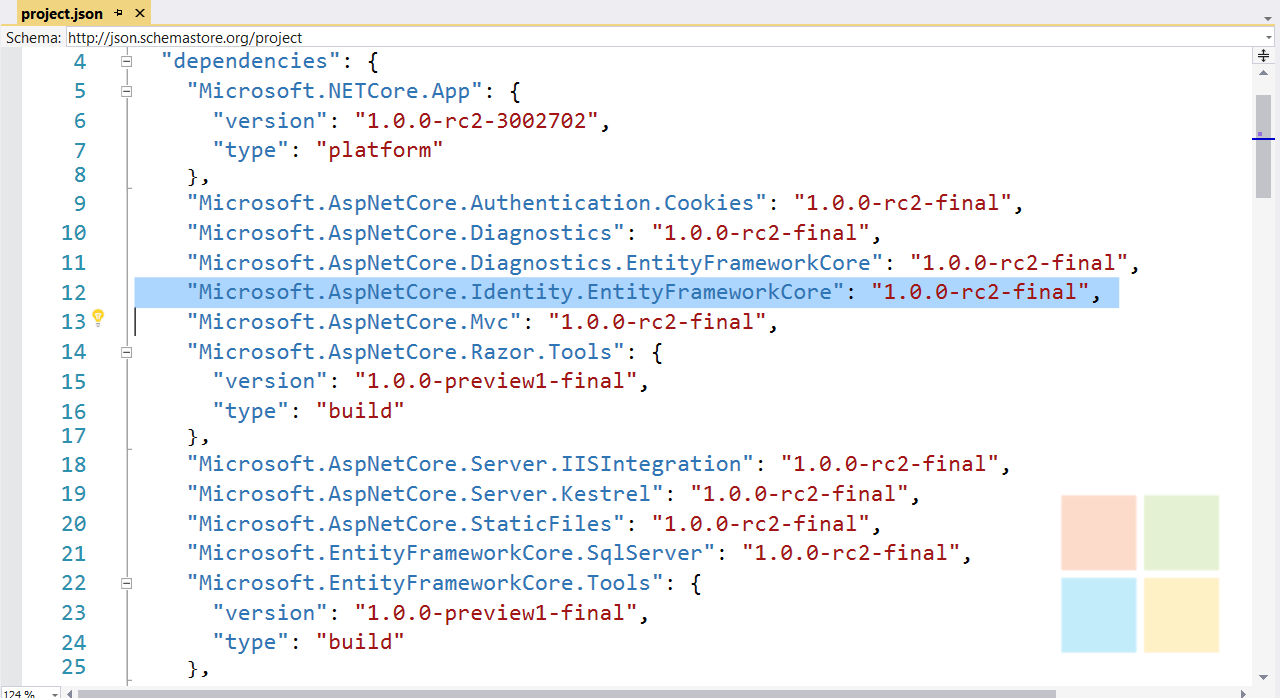
*Создание нового проекта ASP.NET Web Application*

1. В диалоговом окне **New ASP.NET Project** выберите шаблон **Web Application**. Также убедитесь, что параметры **Authentication** установлены как **Individual User Accounts**. Нажмите **OK** для продолжения.



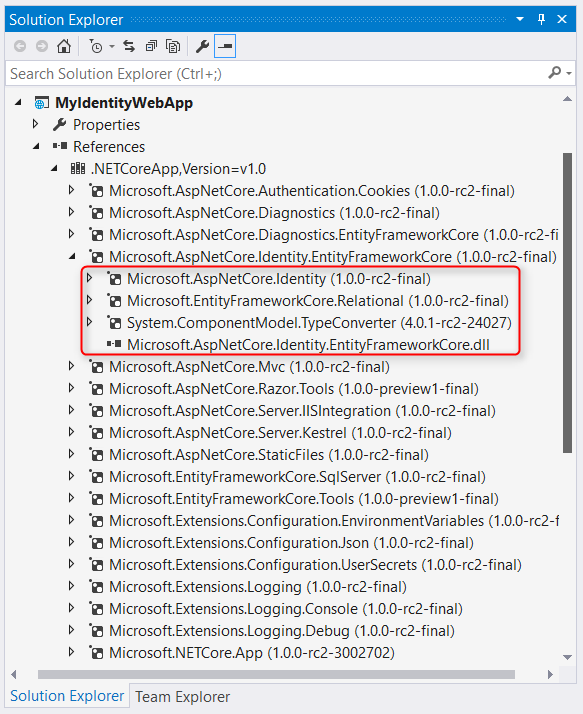
*Создание нового проекта при помощи шаблона Web Application*

1. Как только проект создан, откройте файл *project.json* file и разместите пакет *Microsoft.AspNet.Identity.EntityFramework*. Этот пакет имеет исполнение **Entity Framework** **ASP.NET Identity,** которое перенесет данные и схемы ASP.NET Identity на SQL Server.



*Пакет Microsoft.AspNet.Identity.EntityFramework*

1. Раскройте узел ссылки в Solution Explorer и затем раскройте пакет *Microsoft.AspNet.Identity.EntityFramework* внутри *DNX 4.5.1*. Обратите внимание, что это зависит от *Microsoft.AspNet.Identity,* которая является основной ссылкой для сборки системы ASP.NET Identity. Эта сборка содержит базовый набор ASP.NET Identity.



*Пакет зависимостей Microsoft.AspNet.Identity.EntityFramework*

1. Откройте файл *Startup.cs* и найдите метод **ConfigureServices**. В этом методе сервисы Identity сконфигурированы следующим кодом.

public void ConfigureServices(IServiceCollection services)

{

// ...

services.AddIdentity<ApplicationUser, IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

// ...

}

1. В том же файле найдите метод **Configure**, который будет вызван после того, как метод **ConfigureServices** вызывается во время выполнения потока запуска. В этом методе **ASP.NET Identity** включена для приложения, когда метод **UseIdentity** вызывается. Это добавляет проверку подлинности на основе куки на конвейере запросов.

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)

{

// ...

app.UseIdentity();

// ...

}

1. Откройте файл *AccountController.cs*, расположенный в папке **Controllers** и найдите действие **Register** с атрибутом **HttpPost**. Это действие вызывает службу **UserManager** для создания и пользовательского входа на основе информации **RegisterViewModel.**

[HttpPost]

[AllowAnonymous]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Register(RegisterViewModel model, string returnUrl = null)

{

ViewData["ReturnUrl"] = returnUrl;

if (ModelState.IsValid)

{

var user = new ApplicationUser { UserName = model.Email, Email = model.Email };

var result = await \_userManager.CreateAsync(user, model.Password);

if (result.Succeeded)

{

// ...

await \_signInManager.SignInAsync(user, isPersistent: false);

// ...

1. Найдите действие **Login** при помощи аттрибутива **HttpPost**. Это действие осуществляет вход пользователя при помощи метода **PasswordSignInAsync** сервиса **SignInManager**.

[HttpPost]

[AllowAnonymous]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)

{

ViewData["ReturnUrl"] = returnUrl;

if (ModelState.IsValid)

{

var result = await \_signInManager.PasswordSignInAsync(model.Email, model.Password, model.RememberMe, lockoutOnFailure: false);

if (result.Succeeded)

{

// ...

1. Теперь найдите действие **LogOff**. Это действие вызывает метод S**ignOutAsync** сервиса **SignInManager,** который очищает требования пользователя, сохраненные в куки.

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> LogOff()

{

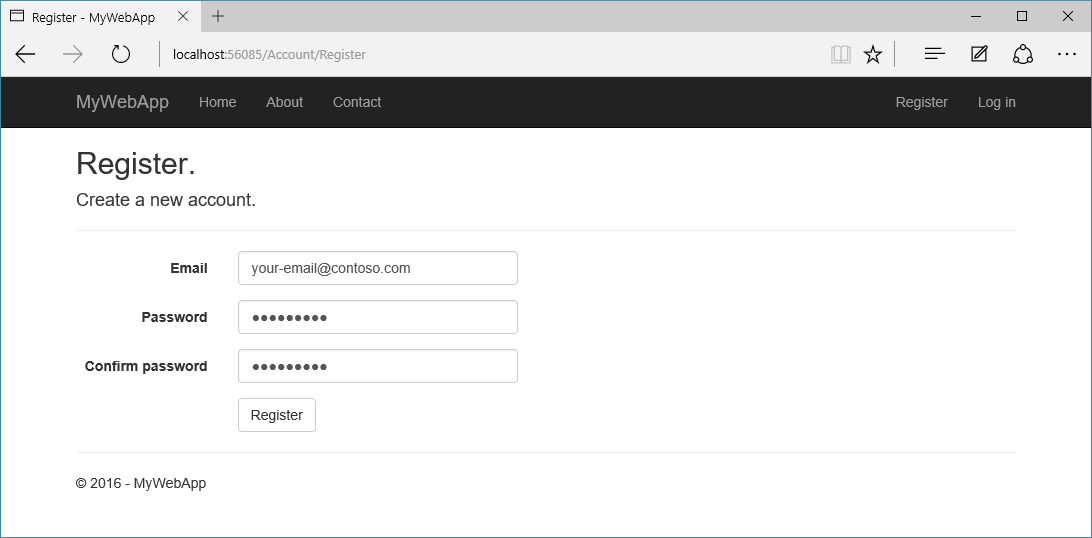
await \_signInManager.SignOutAsync();

\_logger.LogInformation(4, "User logged out.");

return RedirectToAction(nameof(HomeController.Index), "Home");

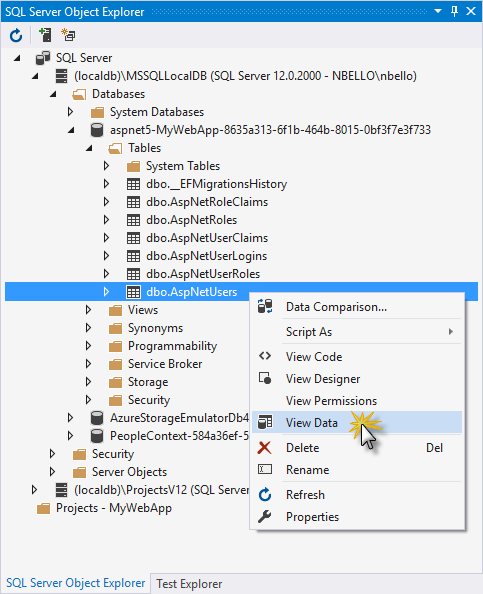
}

1. Выполните решение и создайте нового пользователя, нажав на **Register**, чтобы увидеть систему **ASP.NET Identity** в действии. Вы можете отлаживать различные действия в **AccountController**.

[](https://github.com/Microsoft-Build-2016/CodeLabs-WebDev/blob/master/Module2-AspNetCore/Images/account-register-view.png?raw=true)

*Вид Account register*

1. После регистрации вашего первого пользователя, вы увидите сообщение об ошибке с предложением применить существующие миграции. Нажмите кнопку **Apply Migrations**. Теперь вы увидите, что вы вошли в систему как новый пользователь.
2. Остановите приложение и исследуйте базу данных, перейдя к **(localdb)MSSQLLocalDB/Databases/aspnet5-MyWebApp-/Tables** в **SQL Server Object Explorer**. Щелкните правой кнопкой мыши таблицу **dbo.AspNetUsers** и выберите **View Data**, чтобы увидеть свойства пользователя, которые вы создали.

[](https://github.com/Microsoft-Build-2016/CodeLabs-WebDev/blob/master/Module2-AspNetCore/Images/viewing-user-data-in-sql-server-explorer.png?raw=true)

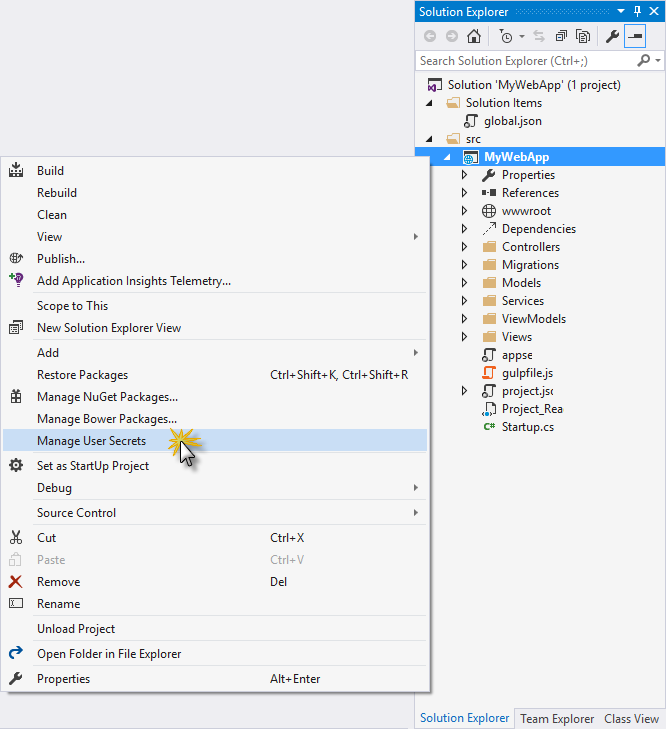
*Просмотр User Data в SQL Server Object Explorer*

**Задание 2 - Включение аутентификации с помощью внешних провайдеров**

ASP.NET Core поддерживает вход с использованием учетных данных OAuth 2.0 от внешнего провайдера аутентификации, таких как Facebook, Twitter, LinkedIn, Microsoft или Google. Включение учетных данных для входа в социальные веб-сайты обеспечивает значительное преимущество, потому что миллионы пользователей уже имеют учетные записи у этих внешних поставщиков. Эти пользователи могут быть более склонны подписаться на ваш сайт, если у них нет необходимости создавать и запоминать новый набор учетных данных.

В этой задаче вы создадите приложение Facebook и настроите ваш проект веб-приложений, который позволяет пользователям войти в систему с помощью аккаунта Facebook в качестве внешнего поставщика.

1. В вашем браузере выберите <https://developers.facebook.com/apps> и войдите при помощи учетных данных Facebook. Если вы еще не зарегистрированы как разработчик Facebook, кликните **Register as a Developer** и следуйте инструкциям по регистрации.
2. На домашней странице **Facebook for developers** добавьте новое приложение, кликнув на **Add a New App** и выбрав **Website** из выборов платформы.
3. На странице **Quick Start for Website** выберите **Skip and Create App ID**.
4. Задайте отображаемое имя, например, *ASP.NET Social Logins*, и выберите категорию, например, *Business*, а затем нажмите **Create App ID**.
5. В секции **Basic** страницы **Settings** кликните **Add Platform,** чтобы указать, что вы добавляете вебсайт приложение.
6. Выберите **Website** из вариантов платформы, добавьте URL вашего сайта (например, *https://localhost:44300/*) и кликните внизу на **Save Changes**.
7. Запишите ваш **App ID** и ваш **App Secret,** чтобы вы могли добавить их в ASP.NET Core Web site позже.
8. Переключитесь назад на **Visual Studio**, правой кнопкой мыши кликните на проект **MyWebApp** и выберите **Manage User Secrets**.

[](https://github.com/Microsoft-Build-2016/CodeLabs-WebDev/blob/master/Module2-AspNetCore/Images/selecting-manage-user-secrets.png?raw=true)

*Выбор Manage User Secrets*

1. В файле *secrets.json* добавьте следующий код, заменив заполнители значениями, которые вы получили от **Facebook**.

{

"Authentication": {

"Facebook": {

"AppId": "<your-app-id>",

"AppSecret": "<your-app-secret>"

}

}

}

1. Откройте файл *project.json* и добавьте пакет **Microsoft.AspNet.Authentication.Facebook** как зависимость

"dependencies": {

...

"Microsoft.AspNetCore.Authentication.Facebook": "1.0.0-rc2-final",

},

1. Откройте файл *startup.cs* и добавьте связующее ПО Facebook в метод **Configure,** как это показано в следующем фрагменте кода.

(Code Snippet - *ASPNETCore - Ex4 - UseFacebookAuthentication*)

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)

{

// ...

app.UseIdentity();

app.UseFacebookAuthentication(new FacebookOptions

{

AppId = Configuration["Authentication:Facebook:AppId"],

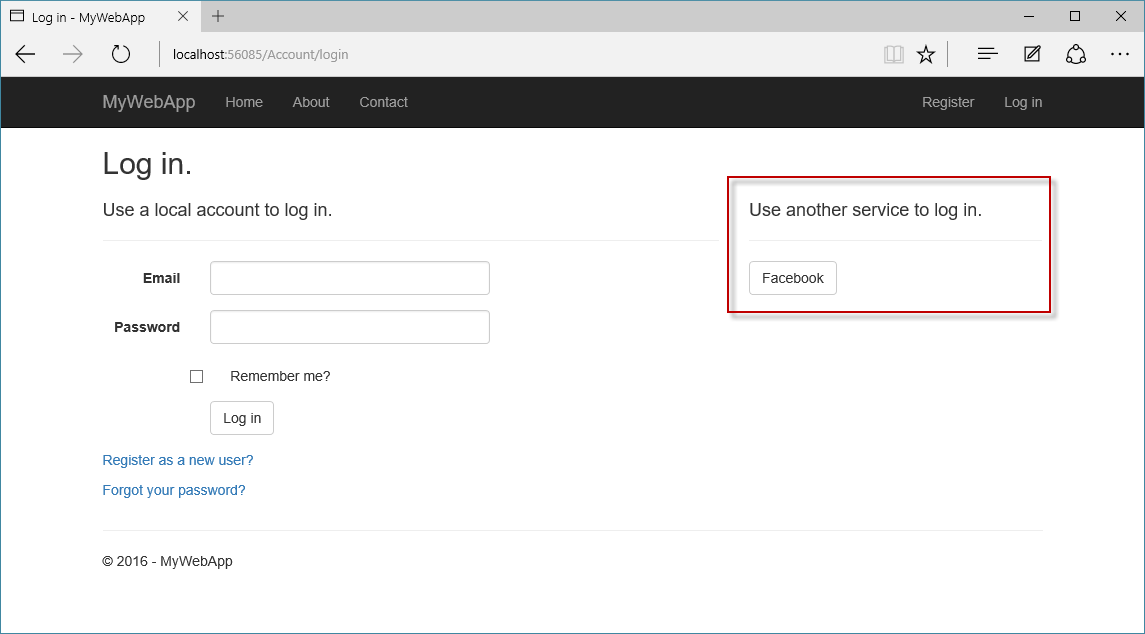
AppSecret = Configuration["Authentication:Facebook:AppSecret"]

});

// ...

}

1. Запустите приложение и перейдите на страницу **Log in**, где вы увидите кнопку **Facebook**.

[](https://github.com/Microsoft-Build-2016/CodeLabs-WebDev/blob/master/Module2-AspNetCore/Images/log-in-page-with-facebook.png?raw=true)

*Страница входа с кнопкой Facebook*

**Итоги**

Завершив этот модуль, вы научились:

* Работать со статическими файлами
* Работать с маршрутизацией, используя MVC
* Создавать заказное связующее ПО
* Добавлять Аутентификацию к вашим веб-приложениям

**Примечание:** воспользуйтесь подпиской на [Visual Studio Dev Essentials](https://www.visualstudio.com/en-us/products/visual-studio-dev-essentials-vs.aspx), чтобы получить все, что нужно для создания и развертывания вашего приложения на любой платформе.